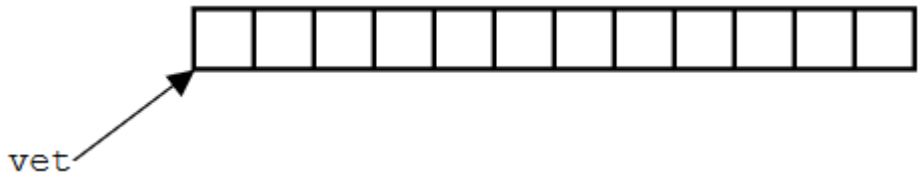
Listas Encadeadas Simples

Prof. Leandro Colevati

Introdução

Para representarmos um grupo de dados, podemos usar um vetor.

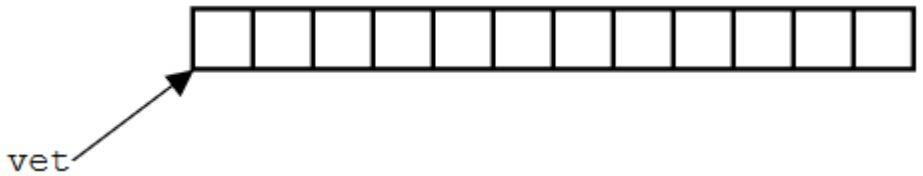
O vetor é a forma mais primitiva de representar diversos elementos agrupados. Ao declararmos um vetor, reservamos um espaço contíguo de memória para armazenar seus elementos, conforme ilustra a figura abaixo.



Introdução

O fato de o vetor ocupar um espaço contíguo na memória nos permite acessar qualquer um de seus elementos a partir do ponteiro para o primeiro elemento. De fato, o símbolo vet, após a declaração acima, como já vimos, representa um ponteiro para o primeiro elemento do vetor, isto é, o valor de vet é o endereço da memória onde o primeiro elemento do vetor está armazenado. De posse do ponteiro para o primeiro elemento, podemos acessar qualquer elemento do vetor através do operador de indexação vet[i].

Dizemos que o vetor é uma estrutura que possibilita acesso randômico aos elementos, pois podemos acessar qualquer elemento aleatoriamente.



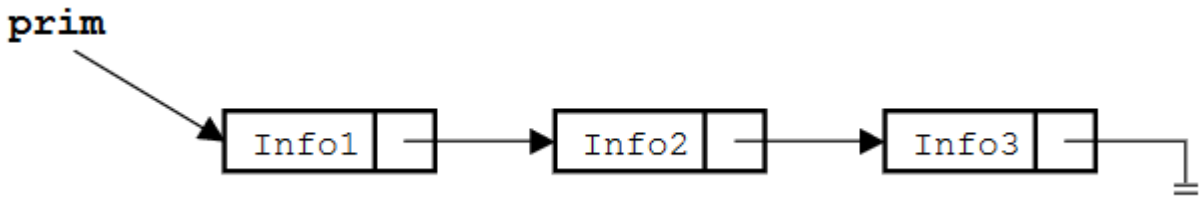
Introdução

No entanto, o vetor não é uma estrutura de dados muito flexível, pois precisamos dimensioná-lo com um número máximo de elementos. Se o número de elementos que precisarmos armazenar exceder a dimensão do vetor, teremos um problema, pois não existe uma maneira simples e barata (computacionalmente) para alterarmos a dimensão do vetor em tempo de execução. Por outro lado, se o número de elementos que precisarmos armazenar no vetor for muito inferior à sua dimensão, estaremos sub-utilizando o espaço de memória reservado.

A solução para esses problemas é utilizar estruturas de dados que cresçam à medida que precisarmos armazenar novos elementos (e diminuam à medida que precisarmos retirar elementos armazenados anteriormente). Tais estruturas são chamadas dinâmicas e armazenam cada um dos seus elementos usando alocação dinâmica.

Definição

Numa lista encadeada, para cada novo elemento inserido na estrutura, alocamos um espaço de memória para armazená-lo. Desta forma, o espaço total de memória gasto pela estrutura é proporcional ao número de elementos nela armazenado. No entanto, não podemos garantir que os elementos armazenados na lista ocuparão um espaço de memória contíguo, portanto não temos acesso direto aos elementos da lista. Para que seja possível percorrer todos os elementos da lista, devemos explicitamente guardar o encadeamento dos elementos, o que é feito armazenando-se, junto com a informação de cada elemento, um ponteiro para o próximo elemento da lista.



Definição

Numa lista encadeada, para cada novo elemento inserido na estrutura, alocamos um espaço de memória para armazená-lo. Desta forma, o espaço total de memória gasto pela estrutura é proporcional ao número de elementos nela armazenado. No entanto, não podemos garantir que os elementos armazenados na lista ocuparão um espaço de memória contíguo, portanto não temos acesso direto aos elementos da lista. Para que seja possível percorrer todos os elementos da lista, devemos explicitamente guardar o encadeamento dos elementos, o que é feito armazenando-se, junto com a informação de cada elemento, um ponteiro para o próximo elemento da lista.

A estrutura consiste numa sequência encadeada de elementos, em geral chamados de nós da lista. A lista é representada por um ponteiro para o primeiro elemento (ou nó).

Do primeiro elemento, podemos alcançar o segundo seguindo o encadeamento, e assim por diante. O último elemento da lista aponta para NULL, sinalizando que não existe um próximo elemento.

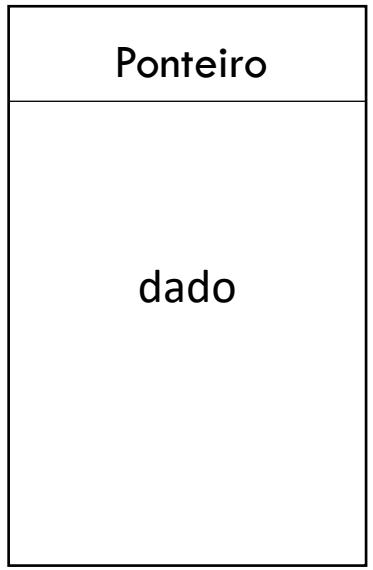
Introdução

- Operações Básicas:
  - Teste de lista vazia;
  - Criação da lista;
  - Adicionar um elemento no início da lista;
  - Adicionar um elemento no fim da lista;
  - Adicionar um elemento em qualquer lugar da lista;
  - Remover um elemento do início da lista;
  - Remover um elemento no fim da lista;
  - Remover um elemento em qualquer lugar da lista;
  - Acesso aos elementos da lista.
    - Verificar elemento de uma posição da lista
    - Tamanho

Alocação Dinâmica

- Considere a definição do tipo Lista abaixo:

```
class No {
   tipo    dado;
   No      próximo; //Ponteiro
}
```



Alocação Dinâmica

- Considere a definição do tipo Lista abaixo:
  - Ponteiro Primeiro → NULL

Alocação Dinâmica

- Teste de lista vazia:

```
No primeiro;
booleano listaVazia() {
    se (primeiro == nulo) {
        retorne verdadeiro;
    } senão {
        retorne falso;
    }
}
```

Alocação Dinâmica

- Adicionar elemento no início:

```
No primeiro;
void addFirst(tipo valor) {
        No elemento = new No();
        elemento.dado = valor;
        elemento.proximo = primeiro;
        primeiro = elemento;
}
```

Alocação Dinâmica

- Tamanho da lista:

```
No primeiro;
int size() {
        int cont = 0;
        se (listaVazia() == falso) {
                No auxiliar = primeiro;
                enquanto (auxiliar != nulo) {
                   cont = cont + 1;
                   auxiliar = auxiliar.proximo;
                }
        }
        retorne cont;
}
```

Alocação Dinâmica

- Retornar um Nó (Operação auxiliar):

```
No primeiro;
private No getNo() {
        se (pos < 0) {
            exceção(“Posição inválida”);
        }
        No auxiliar = primeiro;
        int cont = 0;
        enquanto (cont < pos) {
            auxiliar = auxiliar.proximo;
            cont = cont + 1;
        }
        retorne auxiliar;
}
```

Alocação Dinâmica

- Adicionar elemento no fim:

```
No primeiro;
void addLast(tipo valor) {
        int tamanho = size();
        se (listaVazia() == verdadeiro) {
            exceção(“Lista Vazia”);
        }
        No elemento = new No();
        elemento.dado = valor;
        elemento.proximo = nulo;
        No ultimo = getNo(tamanho - 1);
        ultimo.proximo = elemento;
}
```

Alocação Dinâmica

- Adicionar elemento em qualquer posição válida:

```
No primeiro;
void addPosition(tipo valor, int pos) {
        Int tamanho = size();
        se (pos < 0 || pos > tamanho) {
            exceção(“posição inválida”);
        }
        No elemento = new No();
        elemento.dado = valor;
        se (pos == 0) {
            addFirst(valor);
        } else {
            No anterior = getNo(posAnt);
            elemento.proximo = anterior.proximo;
            anterior.proximo = elemento;
        }
}
```

Alocação Dinâmica

- Remover elemento do início:

```
No primeiro;
void removeFirst() {
        se (listaVazia() == verdadeiro) {
            exceção(“Lista Vazia”);
        }
        primeiro = primeiro.proximo;
}
```

Alocação Dinâmica

- Remover elemento do fim:

No primeiro;

```
void removeLast() {
        se (listaVazia() == verdadeiro) {
            exceção(“Lista Vazia”);
        }
        int tamanho = size();
        No penultimo = getNo(tamanho – 2);
        penultimo.proximo = null;
}
```

Alocação Dinâmica

- Remover elemento do qualquer posição válida:

No primeiro;

```
void removePosition(int pos) {
        Int tamanho = size();
        se (pos < 0 || pos > tamanho) {
            exceção(“posição inválida”);
        }
        se (listaVazia() == verdadeiro) {
            exceção(“Lista Vazia”);
        }
        se (pos == 0) {
            removeFirst();
        } else {
            No anterior = getNo(pos – 1);
            No atual = getNo(pos);
            anterior.proximo = atual.proximo;
        }
}
```

Alocação Dinâmica

- Acessando elementos da Lista
  - Como estamos usando uma lista simplesmente encadeada podemos acessar todos os elementos da fila, a partir do primeiro.

Alocação Dinâmica

- Verificando o elemento de uma posição:

```
No primeiro;
tipo get(int pos) {
    se (listaVazia() == verdadeiro) {
        exceção(“Lista Vazia");
    }
    int tamanho = size();
    se (pos < 0 || pos > tamanho - 1) {
        exceção(“Posição inválida");
    }
    int cont = 0;
    No auxiliar = primeiro;
    enquanto (cont < pos) {
        auxiliar = auxiliar.proximo;
        cont++;
    }
    return auxiliar.dado;
}
```

Alocação Dinâmica

- Exemplo(Lista de Strings):

```
class exemplo {
        void main(String[] args) {
            Lista l = new Lista()
            booleano vazia = l.listaVazia();
            escreva(vazia);
            int tamanho = l.size();
            escreva("Tamanho:"+tamanho);
            String valor = l.get(0);
           l.addFirst("C");
           l.addFirst("B");
           l.addFirst("A");
           l.addLast("D");
           l.addPosition("Y", 0);
           l.addPosition("X", 2);
            tamanho = l.size();
            escreva("Tamanho:"+tamanho);
            dado = l.get(0);
            escreva(dado);
            l.addPosition("K", 10);
            l.addPosition("Z", 6);
            l.removeFirst();
            ...
```

...continuação

```
class exemplo {
        void main(String[] args) {
            ...
            dado = l.get(0);
            escreva(dado);
            l.removeLast();
            tamanho = l.size();
            dado = l.get(tamanho – 1);
            escreva(dado);
            l.removePosition(1);
            tamanho = l.size();
            para (int i = 0 ; i < tamanho ; i++) {
               dado = l.get(i);
               escreva(dado);
            }
        }
}
```

Exercício 1

a) Transformar o projeto de em uma biblioteca, gerando o JAR ListaStrings.

b) Ajustar o projeto de aula para criar uma biblioteca de Lista de Inteiros, gerando o JAR ListaInt

Exercício 2

Considerando o projeto que gerou a biblioteca ListaInt no exercício 1, fazer uma modificação para criar a biblioteca SetInt. Set é uma coleção de dados semelhante a uma Lista Encadeada Simples, no entanto, os métodos de adição de um novo elemento precisam ter uma verificação extra, que consiste como segue:

- Um valor não pode ser inserido na lista se já houver um valor igual adicionado na lista

Criar um novo Projeto Java SetInt semelhante ao projeto de Lista de Inteiros, aplicar a verificação de elementos iguais em todos os métodos de adição e criar a biblioteca com o JAR SetInt

Exercício 3

Considerando a biblioteca ListaString criada no exercício 1, fazer uma aplicação Java que simule uma playlist de músicas em um aplicativo gratuito. O tal aplicativo Vitufy, por ser gratuito, a partir de uma lista, só permite a execução das músicas na ordem em que forem inseridas, não permitindo avançar nem retornar.

Criar uma classe PlayerController no Package controller, que tenha alguns métodos:

- adicionaMusica(Lista lista,String musica):void, que receberá uma String no formato: Nome_da_musica;nome_artista;duracao (obrigatoriamente dessa forma) e, se a lista estiver vazia, adicione na primeira posição. A partir do segundo elemento, todas as outras músicas devem entrar no final da lista;
- removeMusica(Lista lista, int posição): void, permitindo que o usuário remova qualquer música da lista. Se a lista estiver vazia, uma exceção deve ser mostrada;
- executaPlaylist(Lista lista): void, que quebrará a String com uma operação de split() e exibirá, do primeiro ao último elemento, no formato [Música: nome da música – Artista: nome artista – Duração: duracao]. Se a lista estiver vazia, uma exceção deverá ser lançada.

  Bônus: Gerar um sleep para simular a música tocando. Por não ser uma classe tipo Thread, deve-se usar Thread.sleep(tempo); para conseguir o mesmo efeito.

No Package view, criar uma classe chamada Principal, e, na sua Main, inicialize uma Lista Encadeada e oferecer ao usuário a opção de inserir ou remover músicas na playlist e executar a playlist. No caso da inserção de músicas, deve-se pedir, ao usuário, que digite separadamente cada campo e monte uma String que será mandada ao método adicionaMusica na classe PlayerController. A aplicação deve ficar ativa até que o usuário digite uma opção de sair.